CSCI 262
Data Structures

17 – Binary Search Trees

Review: Binary Trees
A binary tree is defined recursively:

- = or
- A binary tree is (empty)

- A root node with a left child and a right child, each of which is a binary tree.

Review: Binary Tree Implementation
Just follow the recursive definition to get a simple implementation:

```cpp
template <class T>
class binary_tree_node {
public:
    T data;
    binary_tree_node<T>* left;
    binary_tree_node<T>* right;
};
```

Search Trees
Data structure for holding comparable elements
- Efficient searching, insertion, deletion
- Underlying structure for sets, maps (BSTs)
- Also used in database indexing (B-Trees)

The basic structure:
- Nodes hold unique data values and pointers to child nodes
- Data acts as a partitioning element in the tree:
  - Child nodes/trees to the left of the data element have value less than the data element
  - Child nodes/trees to the right have a value greater than the data element

Binary Search Trees
Here’s a binary search tree (BST):
- Nodes contain strings
- Left child subtree contains only values less than root
- Root value is less than all right child subtree values

In Order Traversal
Visit left subtree, visit root, visit right subtree.

```cpp
template <class T>
void print_inorder(binary_tree_node<T>* root) {
    if (root != NULL) {
        print_inorder(root->left);
        cout << root->data << " ";
        print_inorder(root->right);
    }
}
```

apple banana cherry guava lemon orange peach pear quince
Searching

```cpp
template <class T>
binary_tree_node<T>* search(binary_tree_node<T>* root, T val) {
    if (root == NULL) return NULL;
    if (val == root->data) return root;
    if (val < root->data) return search(root->left, val);
    else return search(root->right, val);
}
```

Example:
`search(root, "cherry")`

Inserting

Where do we insert an item into the tree?

Answer: put it where you expect to find it!

```cpp
template <class T>
void insert(binary_tree_node<T>*& root, T val) {
    if (root == NULL) root = new binary_tree_node<T>(val);
    else if (val < root->info) insert(root->left, val);
    else if (val > root->info) insert(root->right, val);
}
```

Example:
`insert(root, "fig")`

Removing

1. Find the item
2. Detach and delete

3 Cases when node is in tree:
- 1. No children
- 2. One child
- 3. Two children

Removing Case 1: No Child

Example:
`remove(root, "lemon")`
Removing Case 2: One Child

1. Find the item
2. Link child to parent
3. Delete

Example: remove(root, "quince")

Removing Case 3: Two Children

1. Find the item
2. Swap with rightmost item in left subtree (why?)
3. Remove rightmost node in left subtree (Case 1 or 2)

Example: remove(root, "guava")

Removing: Code

```cpp
template <class T>
void remove(binary_tree_node<T>*& root, T val) {
    if (root == NULL) return NULL;
    if (val < root->data) remove(root->left, val);
    else if (val > root->data) remove(root->right, val);
    else {  // item found!
        if (root->left == NULL || root->right == NULL) {
            binary_tree_node<T>* tmp;
            if (root->left == NULL) tmp = root->right;
            else tmp = root->left;
            delete root;
            root = tmp;
        } else {
            binary_tree_node<T>* tmp = root->left;  // find rightmost node
            binary_tree_node<T>* parent = root;     // in left subtree
            while (tmp->right != NULL) {
                parent = tmp;
                tmp = tmp->right;
            }
            root->data = tmp->data;        // copy data to root
            if (parent == root)            // detach and delete rightmost node
                root->left = tmp->left;     // in left subtree
            else
                parent->right = tmp->left;
            delete tmp;
        }
    }
}
```

Base case: item not found, do nothing.
Recursive calls to find item to delete.
Cases 1 and 2, both. Can you see why case 1 is handled by this block?

Practice With BSTs

https://www.cs.usfca.edu/~galles/visualization/BST.html

Analysis

What is the “big-O” complexity of:
- Searching?
- Inserting?
- Removing?

Complexity of Search
Height of Trees

So how high is a tree with \( N \) nodes?

Best case: \( h = \lceil \log_2(N+1) \rceil = O(\log N) \)

Worst case: \( h = N \)

Height-Balanced Trees (AVL)

Again, a recursive definition:

- Left and right subtrees are height-balanced
- Left and right subtrees differ in height by no more than 1

Which of these are height balanced?

Analysis on Balanced BSTs

When trees are balanced:

- Each subtree contains roughly half the nodes
- Each step down the tree roughly halves the problem

Search, insert, delete are all \( O(\log N) \)

Self Balancing BSTs

- Trees become unbalanced through series of inserts and deletes
- Self-balancing: perform \( O(\log N) \) or fewer operations to rebalance after insert, delete
- Examples of self-balancing BSTs:
  - Red-Black trees
  - AVL trees
  - Splay trees

Rotations

We change the balance at a node via a rotation:

This is the right rotation. The left rotation is the mirror image of this one.

If the tree shown was a BST, is the new tree a BST?

Rebalancing Example (AVL)

Removing this node unbalances the tree at \( e \).

A single right rotation restores balance.
Tree Balancing (AVL)

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

### AVL Tree

<table>
<thead>
<tr>
<th>100</th>
<th>80</th>
<th>90</th>
<th>70</th>
<th>10</th>
<th>60</th>
<th>20</th>
<th>15</th>
<th>25</th>
<th>120</th>
<th>110</th>
</tr>
</thead>
</table>

---

Final Words

Why BSTs matter:
- Linux kernel: schedulers, ext3 filesystem, virtual memory, many more (Red-Black trees)
- Ordered set and map types (e.g., C++ STL, Java) (Red-Black trees again!)
- Database indexing (B-trees – not exactly BSTs, but related)
- Filesystem metadata indexing (B-trees or R-B)
- Lurking in your favorite OS?

---

Up Next

- **Friday, November 9**
  - Lab 10, continued
  - APT 4 due
- **Monday, November 12**
  - Midterm review
  - Lab 10 due
- **Wednesday, November 14**
  - Midterm 2 (in class)
- **Friday, November 16**
  - Fun & Games instead of lab (optional)